Text transformations – Draft

Setup

```
import icu
import el_internationalisation as eli
```

Introduction

In its most general sense, text transformations include:

- Case mappings and case-folding,
- Unicode Normalisation,
- Transforms, and the
- Bidirectional algorithm (rendering of a text flow)

Python, including Pandas, approaches to text transformations include:

Transformation type	Python	Pandas
Case operations	str.lower(), str.upper(), str.title()	pandas.Series.str.lower(), pandas.Series.str.upper(), pandas.Series
Casefolding	str.casefold()	pandas.Series.str.casefold()
Normalization	unicodedata.normalize()	pandas.Series.str.normalize()
Transforms	-	-
Bidirectional algorithm	-	-

N.B. I haven't included str.capitalize() or pandas.Series.str.capitalize() since sentence casing is a typesetting operation rather than a Unicode casing operation. Technically str.title() and pandas.Series.str.title() do not conform to the Unicode titlecasing operation, and shouldn't be considering a casing operation in the same sense as the PyICU equivalent.

The above table provides a summary of available text transformations, but this nptebook will concentrate on the `icu.Transliterator()` class.

Casing

The Unicode Standard makes a distinction between *default casing algorithms* and tailorings which may include contextual and language specific tailorings, including:

- Turkish and Azeri casing rules for *dotted capital I* and *dotless small i*.
- Casing rules for retention of a dot when combining marks are applied to the letetr *i*.
- Titlecasing of *IJ* in Dutch.

- Special titlecasing for orthographies that include word initial caseless letters.
- Uppercasing of ß to ẞ.

Casing operations can change the length of a string, they are not necessarily reversible, and can be context and language dependent. Additionally, not all lowercase characters have an uppercase equivalent, so an uppercase string can potentially include both lowercase and caseless letters.

Python provides simple Unicode casing, that is, Python's casing operations are language and locale insensitive.

Lowercasing

Lowercasing is fairly straightforward string operation in Python:

```
el_lexeme = 'ΚΈΝΩΣΙΣ'
print(el_lexeme.lower())
```

```
κένωσις
```

But casing behaviour can differ between simple and full casing support.

The Turkish uppercase letter İ [U+0130 Latin Capital Letter I With Dot Above] lowercases to i [U+0069 Latin Small Letter I] when language sensitive (full) casing is used.

But for language insensitive (simple) casing İ [U+0130 Latin Capital Letter I With Dot Above] is mapped to i [>U+0069 Latin Small Letter I], ◌̇ [U+0307 Combining Dot Above]

```
tr_city = "İstanbul"
print(f'{tr_city}: {eli.codepoints(tr_city)}')
tr_city_lower = tr_city.lower()
print(f'{tr_city_lower}: {eli.codepoints(tr_city_lower)}')
```

```
İstanbul: 0130 0073 0074 0061 006E 0062 0075 006C
i̇stanbul: 0069 0307 0073 0074 0061 006E 0062 0075 006C
```

It is necessary to use PyICU for language sensitive casing.

```
# 1. Create a locale object
loc = icu.Locale("tr_TR")
# 2. Connvert string to an ICU UnicodeString object
us = icu.UnicodeString(tr_city)
# 3. Lowercase string
tr_city_icu_lower = us.toLower(loc)
print(f'{tr_city_icu_lower}: {eli.codepoints(tr_city_icu_lower)}')
```

```
istanbul: 0069 0073 0074 0061 006E 0062 0075 006C
```

It can be collapsed into one line of code:

```
tr_city_icu_lower2 = icu.UnicodeString(tr_city).toLower(icu.Locale("tr_TR"))
print(f'{tr_city_icu_lower2}: {eli.codepoints(tr_city_icu_lower2)}')
```

```
istanbul: 0069 0073 0074 0061 006E 0062 0075 006C
```

Alternatively, it is possible to use ICU's root locale to get language insensitive casing:

```
lang_insensitive = icu.UnicodeString(tr_city).toLower(icu.Locale.getRoot())
print(f'{lang_insensitive}: {eli.codepoints(lang_insensitive)}')
```

```
i̇stanbul: 0069 0307 0073 0074 0061 006E 0062 0075 006C
```

Uppercase

As with lowercasing

Double-click (or enter) to edit

```
de_lexeme = "buße"
print(f'{de_lexeme.upper()} ({eli.cp(de_lexeme, prefix=True)})')
tr_province = "Diyarbakır"
print(f'{tr_province.upper()} ({eli.cp(tr_province, prefix=True)})')
```

```
BUSSE (U+0062 U+0075 U+00DF U+0065)
DIYARBAKIR (U+0044 U+0069 U+0079 U+0061 U+0072 U+0062 U+0061 U+006B U+0131
```

```
de_lexeme_icu = icu.UnicodeString(de_lexeme).toUpper(icu.Locale("de_DE"))
print(f'{de_lexeme_icu}: {eli.codepoints(de_lexeme_icu)}')
```

```
BUSSE: 0042 0055 0053 0053 0045
```

Casefolding

Casefolding, on the other hand, does not transform text into a specific case, rather it removes case distinctions from strings that are being compared.

- Casefolding is language and locale insensitive
- It does not preserve normalization forms
- Length of string may change
- Context dependent casing does not occur
- Lowercase mapping is used for most characters, but uppercase mapping is used for some

some.

> Case folding is related to case conversion. However, the main purpose of case folding is to contribute to caseless matching of strings, whereas the main purpose of case conversion is to put strings into a particular cased form.
> *Unicode Standard Version 15.0*, Default Case Folding

Unicode normalisation

Each Unicode character can have one ro more canonically equivalent forms. If we look at the letters a, á, and ậ:

```
a = eli.canonical_equivalents_str("a")
a_acute = eli.canonical_equivalents_str("á")
a_circumflex_dotbelow = eli.canonical_equivalents_str("ậ")

print(f"{len(a)}: {a}")
print(f"{len(a_acute)}: {a_acute}")
print(f"{len(a_circumflex_dotbelow)}: {a_circumflex_dotbelow}")
```

```
1: ['U+0061']
3: ['U+00E1', 'U+0061 U+0341', 'U+0061 U+0301']
5: ['U+1EAD', 'U+00E2 U+0323', 'U+0061 U+0302 U+0323', 'U+1EA1 U+0302', 'U+
```

The letter a (U+0061 Latin Small Letter A) only has one canonically equivalent form.

While the letter á (U+00E1 Latin Small Letter A With Acute) has three canonically equivalent representations, one of which `U+0061 U+0341` uses a deprecated combining diacritic, leaving two canonically equivalent forms: `U+00E1` and `U+0061 U+0301`.

When multiple diacritics are involved, canonical equivalence becomes more complex. The letter ậ (U+1EAD Latin Small Letter A With Circumflex And Dot Below) five canonically equivalent versions.

The Unicode mechanism for handling canonical equivalence is normalisation. With standard string processing it is possible to normalise the string to a prefered form. There are four normalisation forms defined by Unicode, but only two of these should be used with most text.

Earlier we discussed the letter á (U+00E1 Latin Small Letter A With Acute) which has a one codepoint representation `U+00E1` and a two codepoint representation `U+0061 U+0301`. In the first representation a single character consisting of a vowel and diacritic components is represented as a single codepoint. THis is referred to as a precomposed character.

The second sequence consists of the vowel followed by a combining diacritic, ie the diacritic is a character in and of itself. This is refered to as a decomposed sequence.

Unicode *Normalisation Form D (NFD)* will decompose character sequences, then canonically order characters, while Unicode *Normalisation Form C (NFC)* will decompose the character

order characters, while Unicode *Normalisation Form C (NFC)* will decompose the character sequence, canonically order characters, then convert the string to its precomposed representation.

The *unicodedata* module provides a function to normalise Unicode strings:

```
unicodedata.normalize(form, str)
```

It is importnt to note, that the version of Unicode that `unicodedata` supports depends on the version of Python you are using. If you need your Unicode support to be current, then you need to always sue the latest version of Unicode or use a drop-in replacement for `unicodedata` that is kept current.

Drop-in replacements for `unicodedata` that are updated and support the latest Unicode versions:

1. unicodedata2
2. unicodedataplus

```
import unicodedata as ud
vi_grapheme = "\u00E2\u0323"
vi_grapheme_nfc = ud.normalize("NFC", vi_grapheme)
vi_grapheme_nfd = ud.normalize("NFD", vi_grapheme)
print(f'Original string: {vi_grapheme} ({eli.cp(vi_grapheme, prefix=True)})')
print(f'NFC string: {vi_grapheme_nfc} ({eli.cp(vi_grapheme_nfc, prefix=True)})')
print(f'NFD string: {vi_grapheme_nfd} ({eli.cp(vi_grapheme_nfd, prefix=True)})')
```

```
Original string: ậ (U+00E2 U+0323)
NFC string: ậ (U+1EAD)
NFD string: ậ (U+0061 U+0323 U+0302)
```

PyICU provides a generic function for normalisation, it also provides specific functions for each normlisation form.

You first create a `Normalizer2` instance, then use the `normalize()` function on the Normalizer2 instance on the string you wish to normalise.

Generic function:

```
import icu
normalizer = icu.Normalizer2.getInstance(None, form, mode)
normalizer.normalize(str)
```

form: normalisation form has a value of `nfc`, `nfkc`, or `nfkc_cf`. ***mode*:** composition mode, has values of `icu.UNormalizationMode2.COMPOSE` or `icu.UNormalizationMode2.DECOMPOSE`.

Normalisation Form	Form specified	Composition mode

Normalisation Form	Form specified	Composition mode
NFC	nfc	icu.UNormalizationMode2.COMPOSE
NFKC	nfkc	icu.UNormalizationMode2.COMPOSE
NFD	nfc	icu.UNormalizationMode2.DECOMPOSE
NKKD	nfkc	icu.UNormalizationMode2.DECOMPOSE
NFKC_CF	nfkc_cf	icu.UNormalizationMode2.COMPOSE

For NFC normalisation:

```
normalizer1 = icu.Normalizer2.getInstance(None, "nfc", icu.UNormalizationMode2.C
vi_icu_nfc = normalizer1.normalize(vi_grapheme)
print(f'NFC string: {vi_icu_nfc} ({eli.cp(vi_icu_nfc, prefix=True)})')
```

```
    NFC string: ậ (U+1EAD)
```

For NFD:

```
normalizer2 = icu.Normalizer2.getInstance(None, "nfc", icu.UNormalizationMode2.D
vi_icu_nfd = normalizer2.normalize(vi_grapheme)
print(f'NFD string: {vi_icu_nfd} ({eli.cp(vi_icu_nfd, prefix=True)})')
```

```
    NFD string: ậ (U+0061 U+0323 U+0302)
```

Specialised functions:

PyICU provides the following functions to create a Normalizer2 instance:

1. `icu.icu.Normalizer2.getNFCInstance()`
2. `icu.icu.Normalizer2.getNFKCInstance()`
3. `icu.icu.Normalizer2.getNFDInstance()`
4. `icu.icu.Normalizer2.getNFKDInstance()`
5. `icu.icu.Normalizer2.getNFKCCasefoldInstance()`

For NFC:

```
# 1. Create a PyICU ICU NFC Normalizer2 instance
normalizer_nfc = icu.Normalizer2.getNFCInstance()

# 2. Normalize string
vi_icu_nfc = normalizer_nfc.normalize(vi_grapheme)
print(f'NFC string: {vi_icu_nfc} ({eli.cp(vi_icu_nfc, prefix=True)})')
```

```
    NFC string: ậ (U+1EAD)
```

For NFD:

```
# 1. Create a PyICU NFD Normalizer2 instance
```

```
normalizer_nfd = icu.Normalizer2.getNFDInstance()

# 2. Normalize string
vi_icu_nfd = normalizer_nfd.normalize(vi_grapheme)
print(f'NFD string: {vi_icu_nfd} ({eli.cp(vi_icu_nfd, prefix=True)})')
```

```
NFD string: ậ (U+0061 U+0323 U+0302)
```

It is important to note that not all graphemes have a precomposed form, therefore such characters are identical in their NFC and NFD forms). If we take the Thuɔŋjäŋ (Dinka) breathy vowel **ë**:

```
din_vowel = "ë"
print(f'Canonical equivalents: {eli.canonical_equivalents_str(din_vowel)}')

din_nfc = ud.normalize("NFC", din_vowel)
print(f"NFC: {din_nfc} ({eli.cp(din_nfc, prefix=True)})")
din_nfd = ud.normalize("NFD", din_vowel)
print(f"NFD: {din_nfd} ({eli.cp(din_nfd, prefix=True)})")
```

```
Canonical equivalents: ['U+025B U+0308']
NFC: ë (U+025B U+0308)
NFD: ë (U+025B U+0308)
```

The sequence `<U+025B U+0308>` has no canonical equivalents and the NFC and NFD versions of the sequence are identical.

ICU transforms

The icu.Transliterator class provides flexible and comprehensive text transformations using a single API.

It can be used for:

- Casing (uppercase, lowercase, and titlecase),
- CJK Fullwidth/Halfwidth conversions,
- Unicode Normalisation (NFC, NFKC, NFKC_CF, NFD, and NFKD),
- Hex and character name conversions, and
- Transcription and transliteration conversions.

Determining what is supported.

ICU uses transliteration transformations defined in CLDR. The each version of ICU, supports the equivalent version of CLDR, so available transformations will differ form version to version.

The function `icu.Transliterator.getAvailableIDs()` will return an

`icu.StringEnumeration` object which can be iterated through, providing all the supported transformations. Some transformations will be langauge specific, while others will be more genric and apply to a script.

To get a list of transformations involving the Ethiopic scipt:

```
# print(", ".join([*Transliterator.getAvailableIDs()]))

def filter_available_transformations(s):
    return [x for x in list(icu.Transliterator.getAvailableIDs()) if s.lower() i

print(", ".join(filter_available_transformations("ethi")))
```

```
Braille-Ethiopic/Amharic, Cyrillic-Ethiopic/Gutgarts, Cyrl-Ethi/Gutgarts, E
```

Or search for a variant transformation defined by a specific agency:

```
print(", ".join(filter_available_transformations("ALALOC")))
```

```
Ethi-Latn/ALALOC, Ethiopic-Latin/ALALOC, Latin-Ethiopic/ALALOC, Latn-Ethi/A
```

For those transformations that are language specific, it is possible to filter for a specific language, for instance to find transforms available for Uzbek:

```
print(", ".join(filter_available_transformations("uz_")))
```

```
uz_Cyrl-uz/BGN, uz_Cyrl-uz_Latn, uz_Latn-uz_Cyrl, Any-uz_Cyrl, Any-uz_Latn
```

Inbuilt transforms

To use ICU's inbuilt transformations:

1. Create a transliterator instance using `icu.Transliterator.createInstance()`
2. Use the transliterator instance's `transliterate` method on a string

```
name_deva = "नागार्जुन"

# 1. Create a transliterator instance for Devanagari to Latin (ISO 15919)
transformer = icu.Transliterator.createInstance("Devanagari-Latin")

# 2. Transliterate the text
name_latin = transformer.transliterate(name_deva)

print(f'{name_deva}: {name_latin}')
```

```
नागार्जुन: nāgārjuna
```

Double-click (or enter) to edit

The above code will convert नागार्जुन to **nāgārjuna** following the romanisation schema published in ISO 15919. Since Devanagari is unicameral, the romanisation is lowercase. To obtain the transliterated string using sentence casing or title casing, it is necessary to use more complex transformations.

Predefined transformations include:

1. Script to script transliteration
2. Langauge specific transformations
3. Casing operations
4. Normalisation
5. Other text transformations

Script to script transliteration

Script	Transform	Description
Arabic	Arab-Latn	Default transliteration for Arabic script to Latin script.
	Arabic-Latin	Default transliteration for Arabic script to Latin script. Alternative label.

Language specific transliterations

Language	Transform	Description
Amharic	Amharic-Latin/BGN	BGN/PCGN romanization for Amharic language
Arabic	Arabic-Latin/BGN	BGN/PCGN romanization for Arabic language

BGN/PCGN romanization are the conventions used by the United States Board on Geographic Names (BGN) and the Permanent Committee on Geographical Names for British Official Use (PCGN).

Casing, Normalisation, and other transformations

Category	Transform	Description
Casing	Any-Lower	
	Any-Upper	Simple casing
	Any-Title	
	az-Lower	
	az-Upper	Full casing (Azeri)
	az-Title	
	el-Lower	
	el-Upper	Full casing (Greek)
	el-Title	
	lt-Lower	
	lt-Upper	Full casing (Lithuanian)
	lt-Title	

	nl-Title	Full Title casing (Dutch)
	tr-Lower	
	tr-Upper	Full casing (Turkish)
	tr-Title	
CJK transformations	Fullwidth-Halfwidth Halfwidth-Fullwidth	Convert between fullwidth and halfwidth charcaters
	Any-NFC	
	Any-NFKC	
Normalisation	Any-NFD Any-NFKD	Unicode normalisation
	Any-FCD	
	Any-FCC	

Any-Hex Any-Hex/Unicode Any-Hex/Java Any-Hex/C Any-Hex/XML Any-Hex/XML10 Any-Hex/Perl

```
# 1. Create a PyICU Transliterator Instance
transformer_u = icu.Transliterator.createInstance("Any-Hex/Unicode")

# 2. Transliterate the text
name_cp = transformer_u.transliterate(name_deva)
unicode_list = " ".join(["U"+x for x in name_cp.split("U") if x])
print(f'{name_deva}\n{unicode_list}')
```

```
नागार्जुन
U+0928 U+093E U+0917 U+093E U+0930 U+094D U+091C U+0941 U+0928
```

Custom rules

```
icu.Transliterator.createFromRules(label, rules, direction)
```

Where:

label: identifier for the transform.
rules: string containing rules used to build Transliterator instance
direction: direction of transformation, either icu.UTransDirection.FORWARD or icu.UTransDirection.REVERSE

```
wp_title = "Dɛ̈tëicëkäŋ akɔ̈ɔ̈n"
transformer_rules = ':: NFD; :: [\u0308] Remove; :: Title; '
custom_transformer = icu.Transliterator.createFromRules("customDinka", transform
print(custom_transformer.transliterate(wp_title))
```

```
Dɛteicekaŋ Akɔɔn
```

This transform will daisy chain two inbuilt tarnsformations and a custom transformation:

This transform will daisy chain two inbuilt transformations and a custom transformation:

1. Normalised string to NFD
2. Remove combining any combining diareses (U+0308), using ICU's UnicodeSet notation
3. Title case string

Much more complex transformations are possible, and it is possible to create rules that will run a range of text transformations on strings, allowing a range of data cleanup functions.

Registering a transformation

When using a custom Transliterator instance within a web microframework, an API endpoint or other scenarios where code persists, rather than recreating the instance each time, it can be created, registered and then used the same way ICU internal transformations are used.

1. Create a custom Transliterator instance
2. Register instance

Use the following command:

```
icu.Transliterator.registerInstance(instance)
```

Resources

- icu::Transliterator Class Reference (icu4c)
- ICU User guide: Transforms
- Transform Rule Tutorial
- Unicode Locale Data Markup Language (UTS 35): Transforms
- Transformations defined in CLDR

Colab paid products - Cancel contracts here